Deadlocks

References:

1.Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 7

System Model

- For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.
- Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
- By definition, all the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case (i.e. if there is some difference between the resources within a category), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".
- Some categories may have a single resource.
- In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:
  - Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. For example the system calls open( ), malloc( ), new( ), and request( ).
  - Use - The process uses the resource, e.g. prints to the printer or reads from the file.
  - Release - The process relinquishes the resource. so that it becomes available for other processes. For example, close( ), free( ), delete( ), and release( ).
- A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set (and which can only be released when that other waiting process makes progress.)

Necessary Conditions

- There are four conditions that are necessary to achieve deadlock:
  - **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
  - **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
  - **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
  - **Circular Wait** - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[i] is waiting for P[(i + 1) % (N + 1)]. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

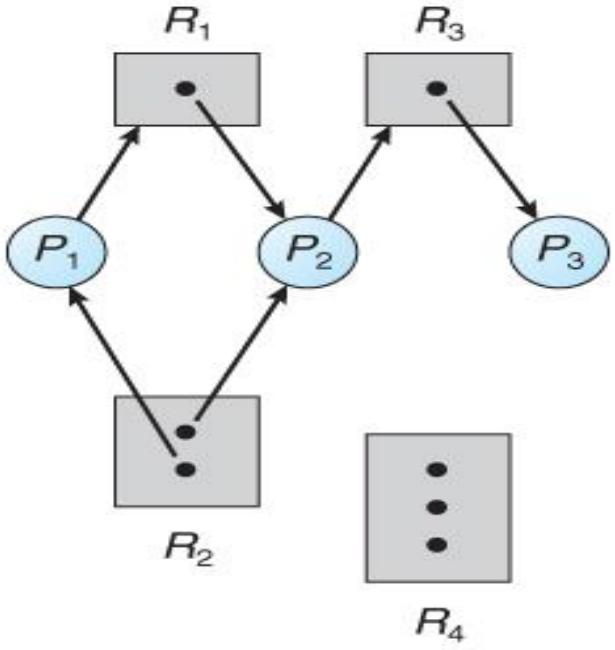
Resource-Allocation Graph

- In some cases deadlocks can be understood more clearly through the use of **Resource-Allocation Graphs**, having the following properties:
  - A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. (E.g. two dots might represent two laser printers.)
  - A set of processes, { P1, P2, P3, . . ., PN }
  - **Request Edges -** A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
  - **Assignment Edges -** A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
  - Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.)

Resource-Allocation Graph

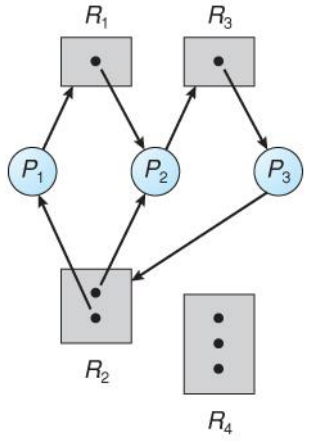
For example:

If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are ***directed*** graphs.) See the example in Figure 7.2 above.

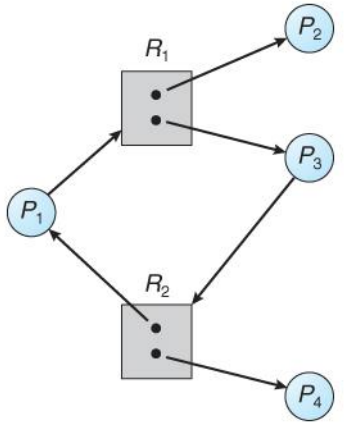


Resource-Allocation Graph

- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one. Consider, for example, Figures 7.3 and 7.4 below:



Resource allocation graph with a deadlock



Resource allocation graph with a cycle but no deadlock

Methods for Handling Deadlocks

Deadlocks can be prevented by preventing at least one of the four required conditions:

7.4.1 Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

7.4.2 Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others.
- There are several possibilities for this:
  - Require that all processes request all resources at one time.
  - This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
  - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request.
    This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
  - Either of the methods described above can lead to starvation if a process requires one or more popular resources.

Deadlock Prevention

Deadlocks can be prevented by preventing at least one of the four required conditions:

7.4.1 Mutual Exclusion

- Shared resources such as read-only files do not lead to deadlocks.
- Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

7.4.2 Hold and Wait

- To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others.
- There are several possibilities for this:
  - Require that all processes request all resources at one time.
  - This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
  - Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request.
    This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
  - Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption

Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.

- One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
- Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
- Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

Circular Wait

- One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
- In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
- One big challenge in this scheme is determining the relative ordering of the different resources.

Deadlock Avoidance

- The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the aforementioned conditions.
- This requires more information about each process, AND tends to lead to low device utilization. (I.e. it is a conservative approach.)
- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation ***state*** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.